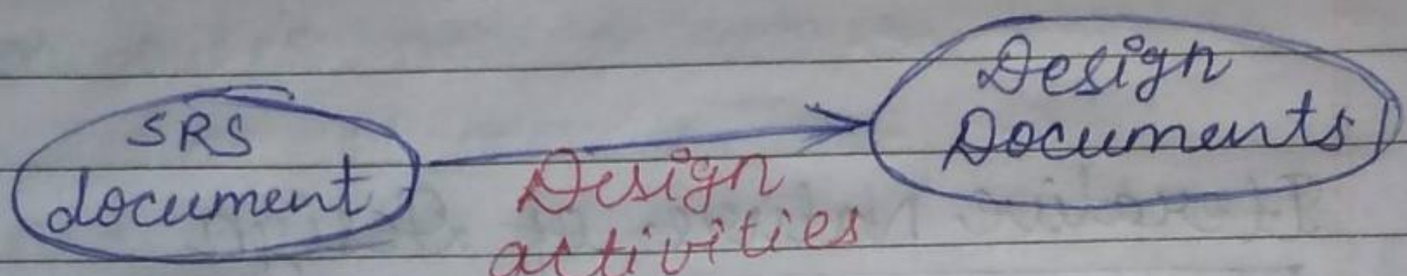
Design Fundamentals :—

What is achieved during design phase?

→ Transformation of SRS document to design document :

— A form easily implementable in some programming language.



Items Designed During Design Phase

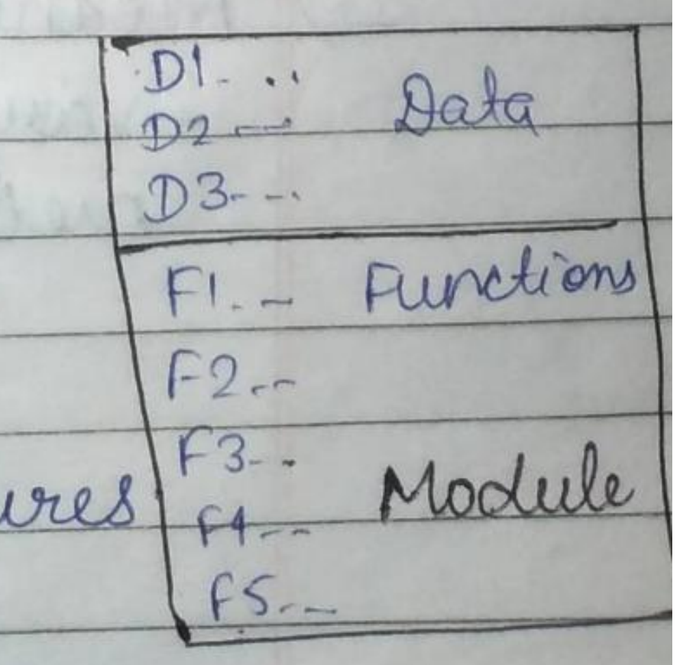
→ Module Structure
→ Control relationship among the modules
— Call relationship or invocation relationship.

→ Interface among different modules,
— data items exchanged among different modules,

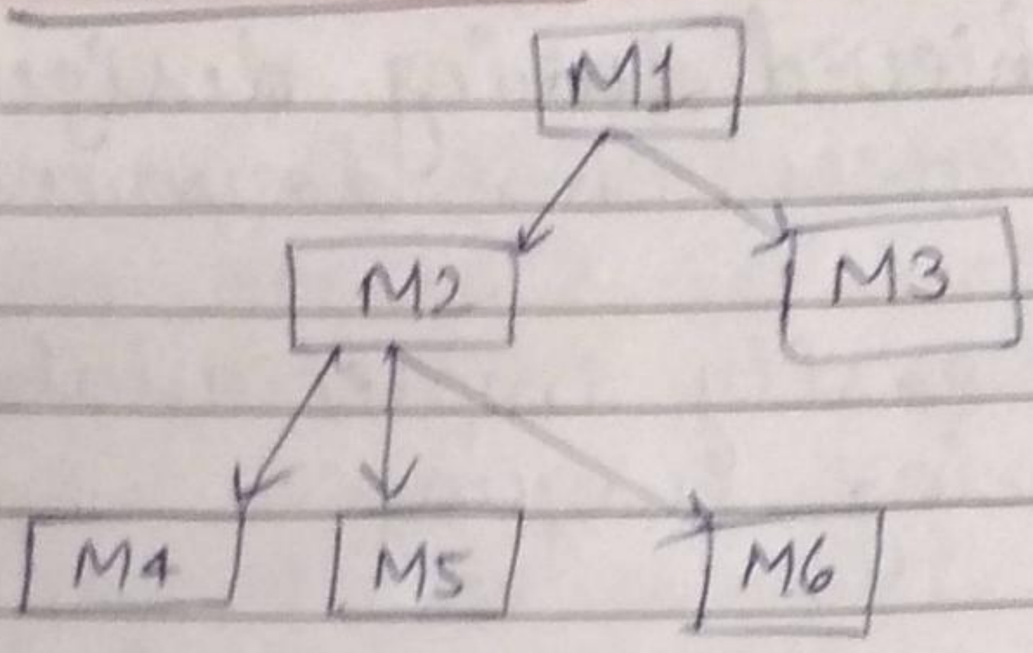
→ Data structures of individual modules,
→ algorithms for individual modules.

Module

→ A module consists of :
— several functions
— associated data structures



Module Structure



Iterative Nature of Design

→ Good software design:
- Seldom arrived through a single step procedure:
- But through a series of steps and Iterations:

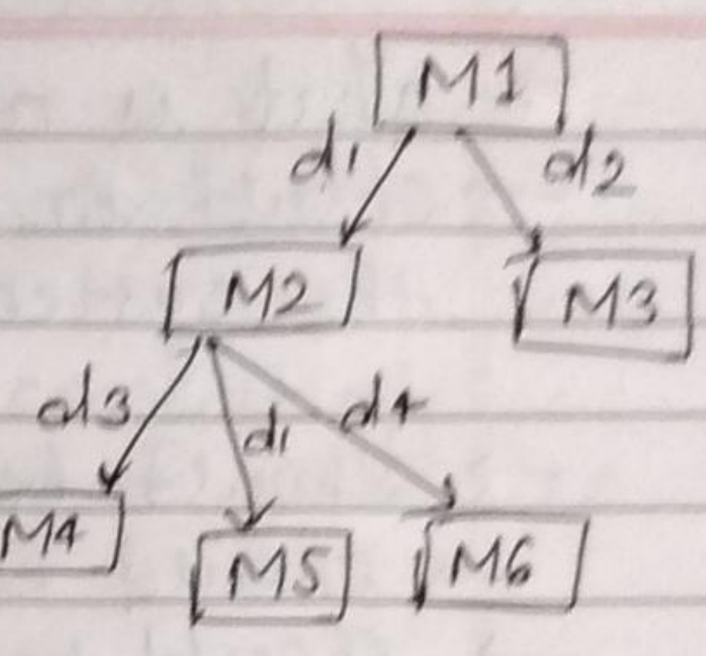
Stages in Design

→ Design activities are usually classified into two stages:
- Preliminary (or high-level) design → architectural design
- Detailed design.

→ Meaning and scope of the two stages:
- Vary considerably from one methodology to another.

High-level design

- Identify:
  - modules
  - control relationships among modules
  - interfaces among modules



→ The outcome of high-level design:
  - program structure, also called s/w architecture.

→ Several notations are available to represent high-level design:
  - Usually a tree-like diagram called structure chart is used.

  - other ~~noth~~ notations:
    - Jackson diagram or Warnier-Orr diagram can also be used.

Detailed Design

→ For each module, design for it:
  - data structure
  - algorithms

→ Outcome of detailed design:
  - module specification.

Date : / /
Page :

** What is a Good Software Design?
→ should implement all functionalities of the system correctly.

→ Should be easily understandable.
→ Should be efficient.
→ Should be easily amenable to change
i.e. easily maintainable.

→ Understandability of a design is a major issue:
- Largely determines goodness of a design:
- a design that is easy to understand:
- also easy to maintain and change.

→ Unless a design is easy to understand,
- tremendous effort needed to maintain it.
- We already know that about 60% effort is spent in maintenance.

→ If the S/W is not easy to understand:
- maintenance effort would increase many times.

Date : / /
Page :

How to Improve Understandability?

→ Use consistent and meaningful names
  – for various design components,

→ Design solution should consist of:
  – A set of well decomposed modules
    (modularity),

→ Different modules should be neatly arranged in a hierarchy:
  – A tree-like diagram.
  – called Layering.

** How are Abstraction and Decomposition principles used in Design?

→ Two principal ways:
  • Modular Design
  • Layered Design

→ Modularity

Modularity is a fundamental attributes of any good design.

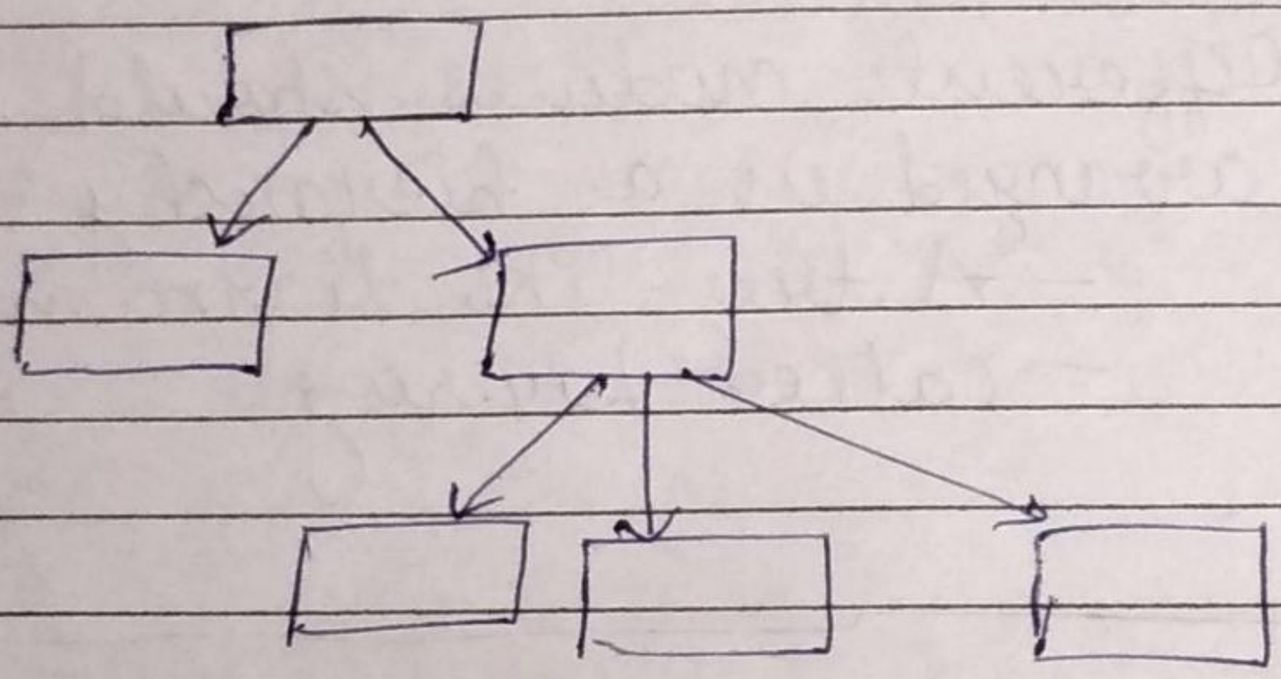
→ Decomposition of a problem cleanly into modules.

→ Modules are almost independent of each other

→ Divide and conquer principle.

→ If modules are independent:
  → Modules can be understood separately
  → Reduces the complexity greatly.

** Layered Design



→ Neat arrangement of modules in a hierarchy means:
  → Low fan-out
  → control abstraction

→ In technical terms, modules should design:
  → High cohesion
  → Low coupling

Cohesion

Cohesion is a measure of:
  → functional strength of a module
  → A cohesive module performs a single task or function.

Date :
Page :

Coupling :

~~A module having high coh~~

→ A measure of the degree of the inter-dependence or interaction between the two modules.

→ A module having high cohesion and low coupling :

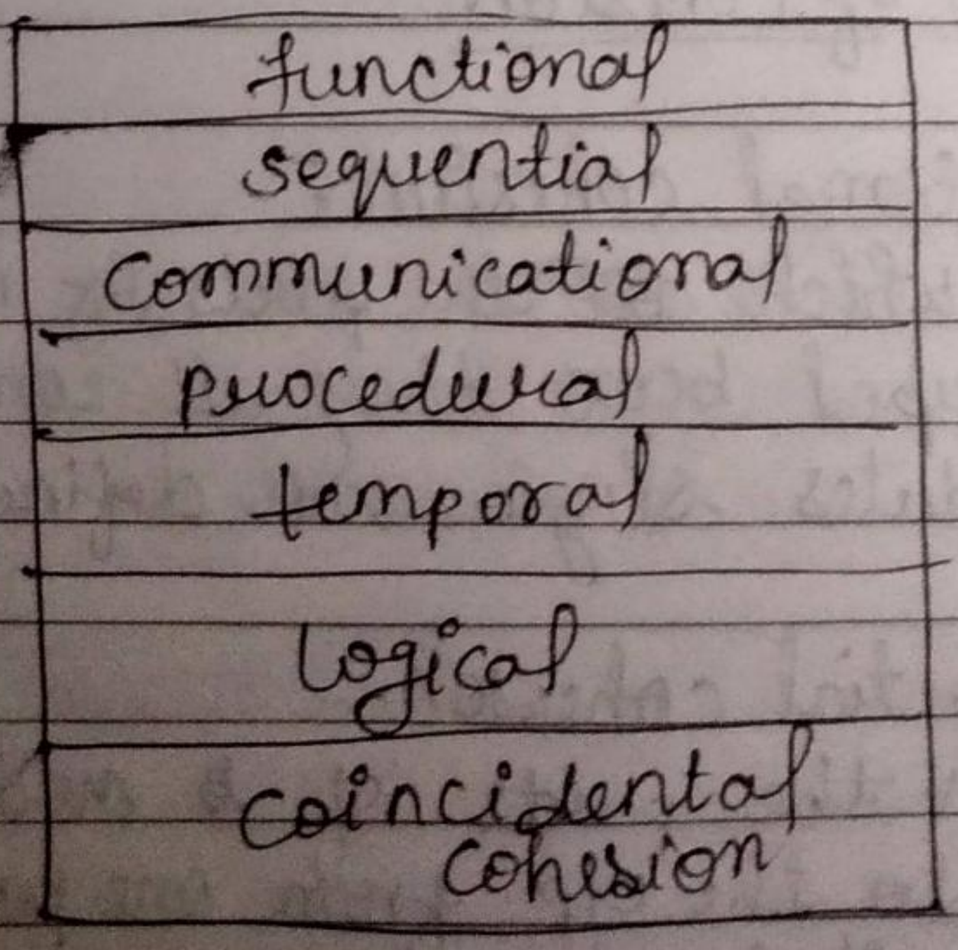
→ functionally independent ~~of~~ of other modules.

→ Advantages of Functional Independence :

(i) Better understandability and good design.
(ii) Complexity of design is reduced
(iii) It reduces error propagation.
(iv) Reuse of modules is possible.

→ There are now ways to quantitatively measure the degree of cohesion and coupling

→ Classification of cohesiveness



high cohesion

⇓

In a good module, various parts having high cohesion is preferable due to its
- reliability
- reusability
- robustness &
- understandability

Low cohesion

⇓

associated with undesirable traits including difficult is maintaining, reusing & understanding

→ Advantages of high cohesion:-

① high cohesion leads to increased module reusuablity.
② System maintainability will increase.
③ Module complexity also reduces.

Note:- The degree of coupling b/w 2 modules depend on their interface complexity.

Types of cohesion

① Functional cohesion:
In which the parts of modules are grouped bcoz they all contribute to the modules single well defined task.

② Sequential cohesion:
When the parts of modules grouped due to the o/p from one part is the i/p to other.

e.g. [sort] → [search] → [Display]

(3) Communication cohesion:
The parts of the module are grouped bcoz they operate on same data.
e.g. the set of func. defined on an array or stack.

(4) Procedural cohesion: The parts of module are grouped bcoz a certain sequence of execution is followed by them.
e.g. Algo for decoding a msg.

(5) logical cohesion:
All the element of module perform similar operations. e.g. error handling, data i/p, data o/p.
An e.g. of logical cohesion –
– A set of print func. to generate an o/p report arranged into a single module.

Date : / /
Page :

Function-Oriented Design

⇓

Focuses on functions

→ It mainly emphasis on "what the system does".

⇓

action

→ FOD, the sys. is comprised of many smaller sub sys known as func.

→ For a FOD design can be represented mathematically or graphically by using
- → Data flow diag. → Process map prog. of sys.
- → Data dictionary → list of diff. data items used in sys
- → Structure charts
- → Pseudo code

→ SA/SD methodology consists of two distinct activities:
- → Structured Analysis (SA)
- → Structured Design (SD)

→ During structured analysis
- → functional decomposition takes place

→ During structured design:
- → module structure is formalized.

→ **Structured Analysis**

→ Transforms a textual problem description into a graphic model.
- → Done using DFDs.

A Date : / /
Page :

→ Cohesion and Coupling:
- Cohesion → functional strength of a module
- Coupling → measure of the degree of the interdependence b/w the module.
  - this has to be low.

→ If functional cohension & data coupling then that is functional indepence.

→ Content diag — level-0 DFD
→ External entities must not represented in higher-level DFD.
i, only in content diag.

→ DFDs graphically represent the results of structured analysis.

→ Structured Analysis Vs. Structured Design.

→ Purpose of structured analysis
→ Capture the detailed structure of the system as the user views it.

→ Purpose of structured design:
→ Arrive at a form that is suitable for implementation in some programming lang.

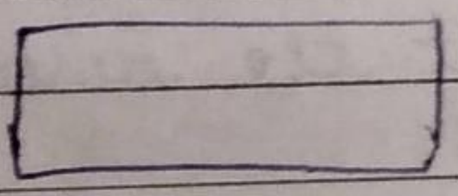
Data Flow Diagram

→ DFD is a hierarchical graphical model:
  → shows the different functions (or processes) of the system and

→ Data interchange among the processes.

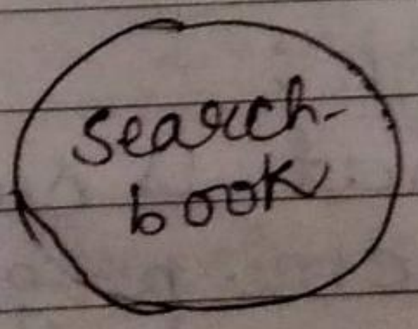
→ It is useful to consider each function as a processing station:
  → Each function consumes some i/p data
  → produces some o/p data.

→ Primitive symbols used for constructing DFDs.

①  → External entity symbol

↗ object

External entity :- Also known as actors, sources or sinks, terminators, are an outside system or process that sends or receives data to and from the diagrammed system.

② 

→ A function such as "search-book" is represented using a circle.

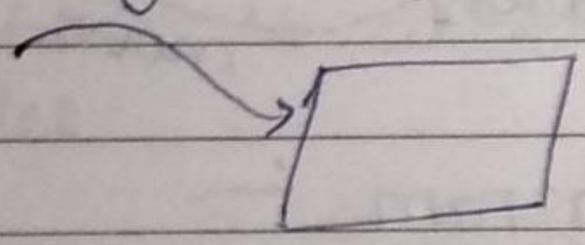
→ This symbol is called a process or bubble or transform.

→ function names should be verbs.

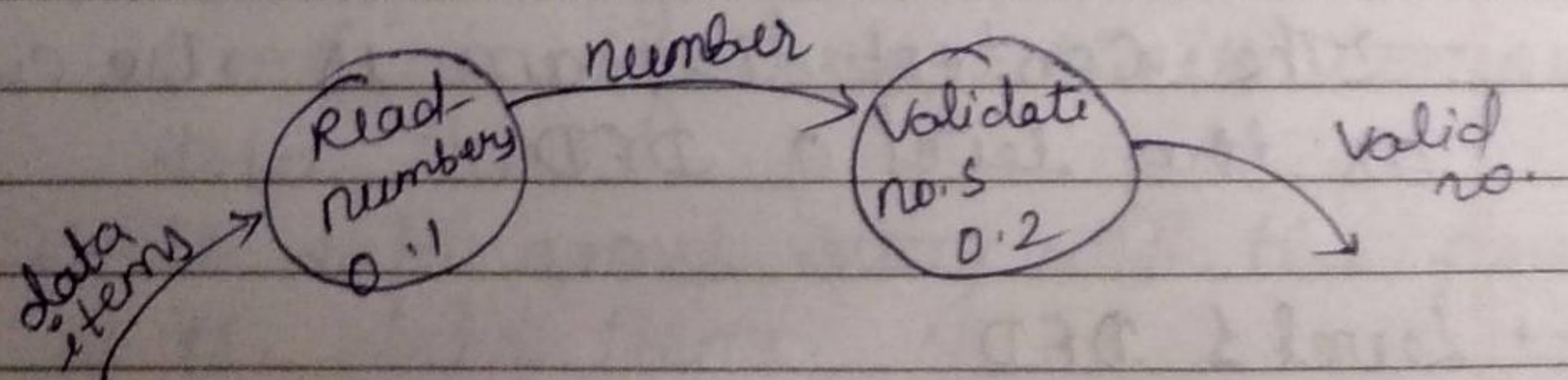
(3) book name ⟶ → A directed line or arc represents data flow in the direction of the arrow.

(4) book-details → represent a logical file
→ A logical file can be:
→ a data structure
→ a physical file on disk

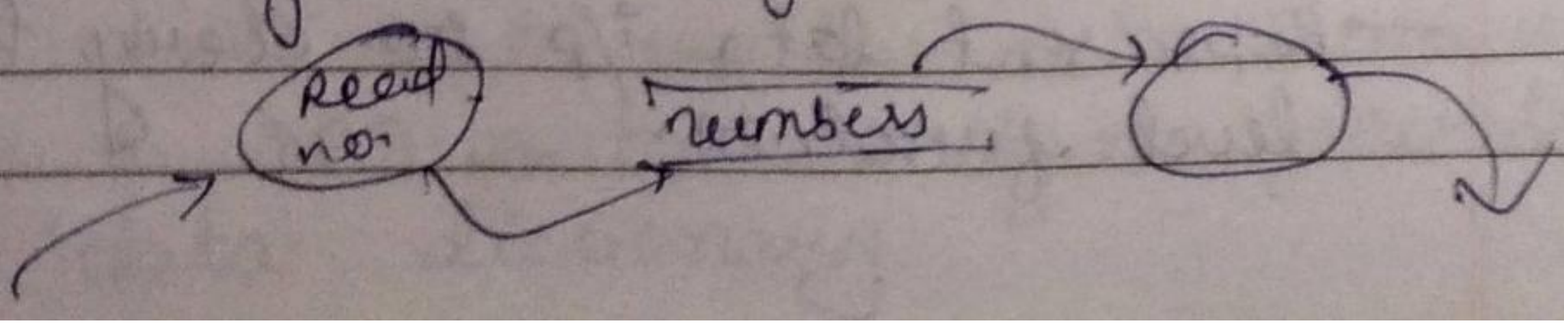
(5) O/P Symbol:
→ O/p produced by the system.



→ Synchronous Operation
→ If two bubbles are directly connected by a data flow arrow: they are synchronous.

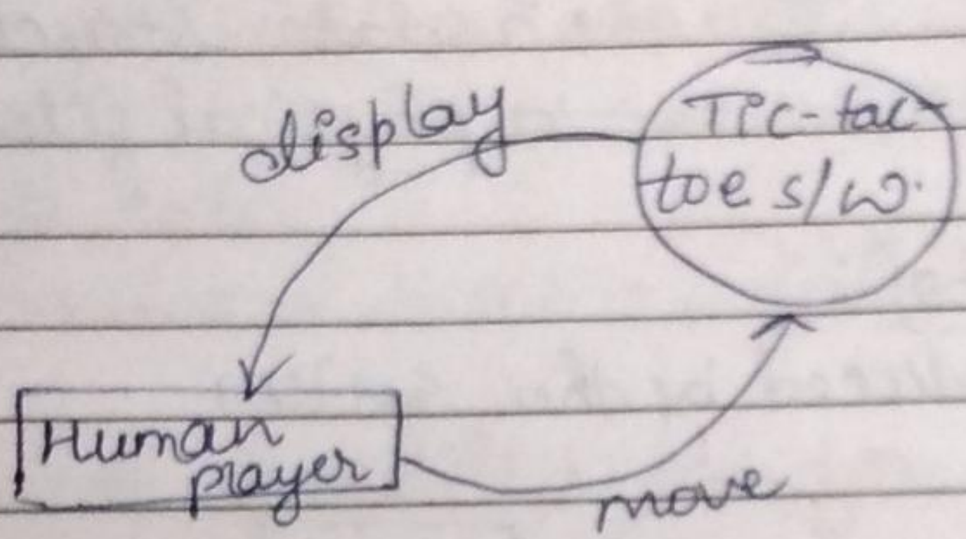


→ Asynchronous operation:
→ If two bubbles are connected via a datastore
→ They are not synchronous.



→ How is Structured Analysis Performed?
→ Initially represent the S/W at the most abstract level:
→ called the content diagram.
→ the entire system is represented as a single bubble.

Tic-tac-toe : Context Diagram



Context Diagram :-

→ A context diagram shows:
→ Data i/p to the system
→ o/p data generated by the system
→ external entities.

→ The context diagram is also called as the level 0 DFD.

→ Level 1 DFD

Examine the SRS document:
→ Represent each high-level function as a bubble.
→ Represent data i/p to every high-level function.

→ Represent data o/p from every high-level function.

→ Higher Level DFDs.

→ Each high-level function is separately decomposed into subfunctions:
  → Identify the subfunctions of the function
  → Identify the data i/p to each subfunction
  → " " " o/p from " "

→ Decomposition of a bubble also called factoring or exploding.
  → Each bubble is decomposed to b/w 3 to 7 bubbles.

→ Data Dictionary :-

A data dictionary lists all data items appearing in a DFD:
  → Definition of all composite data items in terms of their component data items.
  → All data names along with the purpose of the data items.

→ CASE (Computer Aided S/W Engg.) tools come handy:
  → CASE tools capture the data items appearing in a DFD automatically to generate the data dictionary.

Data Definition

Composite data are defined in terms of primitive data items using following operators:

→ + : denotes composition of data items
e.g. a+b.

→ [,,,] : represents selection

→ ( ) : contents inside the bracket represent optional data.

a+(b) → either a or a+b occurs

→ { } : represents iterative data definition

e.g. {name}5 represents five name data.

→ { }* represents zero or more instances of name data.

→ = : repr — equivalence

→ * * : → considered as comment

→ Balancing a DFD

→ Data flowing into or out of a bubble:
→ Must match the data flows at the next level of DFD.
→ means after decomposition DFD should be balanced

→ Numbering of Bubbles:

→ Number the bubbles in a DFD:

→ Numbers help in uniquely identifying any bubble from its bubble number.

→ The bubble at content level:

→ assigned number 0.

→ Bubbles at level 1:

→ Numbered 0.1, 0.2, 0.3 etc.

→ When a bubble numbered x is decomposed.

→ its children bubble are numbered x.1, x.2, x.3, etc.

→ Guidelines for constructing DFDs

Note: (i) Context diagram should represent the system as a single bubble:

(ii) All external entities should be represented in the context diagram.

→ External entities should not appear at any other level of DFD.

(iii) Only 3 to 7 bubbles per diagram should be allowed.

(iv) A DFD does not represent control info:

→ When or in what order different functions are invoked.

→ The conditions under which different functions are not represented.

→ Shortcomings of the DFD Model:

→ DFD models suffer from several shortcomings:

→ DFDs leave ample scope to be imprecise.
  → In a DFD model, we infer about the function performed by a bubble from its label.
  → A label may not capture all the functionality of a bubble.

→ Control information is not represented:
  → For instance, order in which inputs are consumed and outputs are produced is not specified.

→ A DFD does not specify synchronization aspects:

→ For the same problem, several alternative DFD representations are possible.
  → Many times it is not possible to say which DFD representation is superior or preferable.

→ DFD technique does not provide
  → Any clear guidance as to how exactly one should go about decomposing a function.

** Structured Design

→ The aim of structured design
  → Transform the results of structured analysis (i.e. a DFD representation) into a structure chart.

→ A structure chart represents the S/W architecture:
  → Various modules making up the system,
  → Module dependency (i.e. which module calls which others modules)
  → Parameters passed among different modules.

→ Main focus of a structure chart:
  → Define the module structure of a S/W.
  → interaction among different modules
  → Procedural aspects (e.g. how a particular functionality is achieved) are not represented.

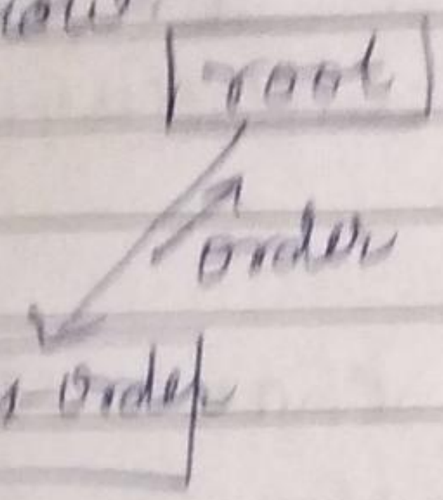
→ Basic Building blocks of structure chart

→ Rectangular box:

[Process-order] → represents a module

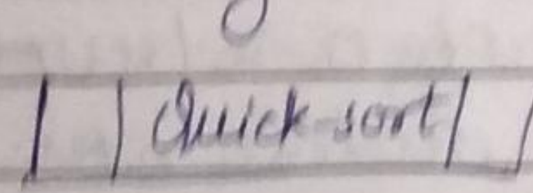
→ Arrow
  → during execution control is passed from one module to the other in the direction of the arrow.

→ Data flow Arrows
→ represent data passing from one module to another in the direction of the arrow.

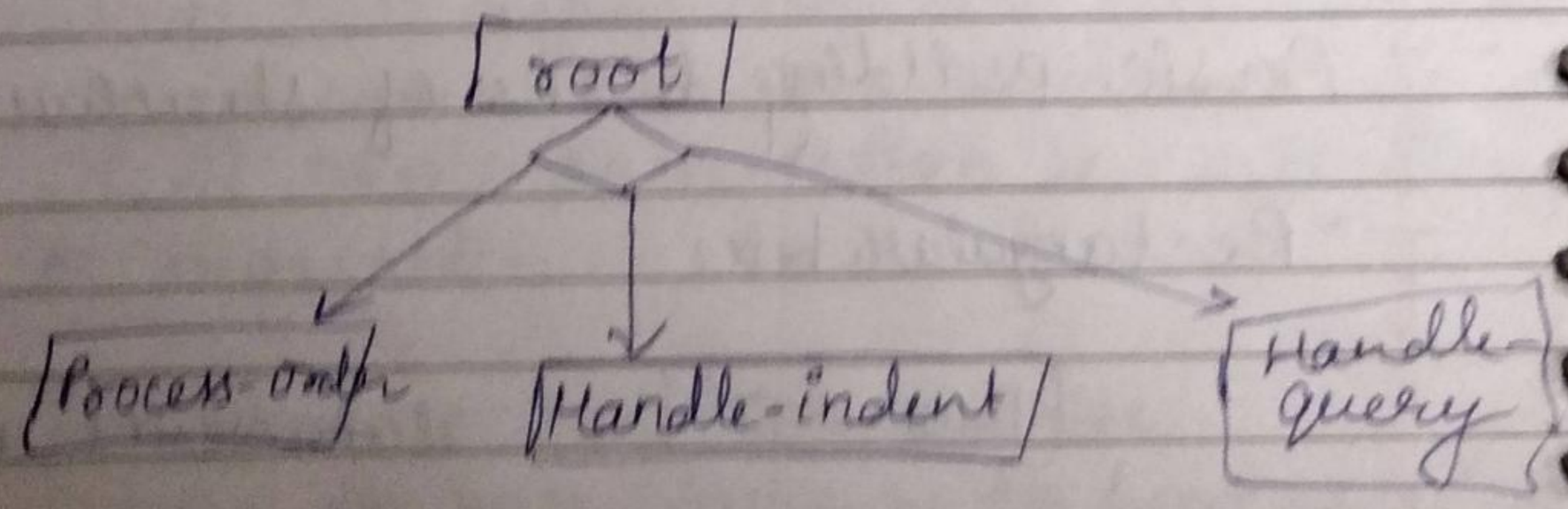


→ Library Modules:

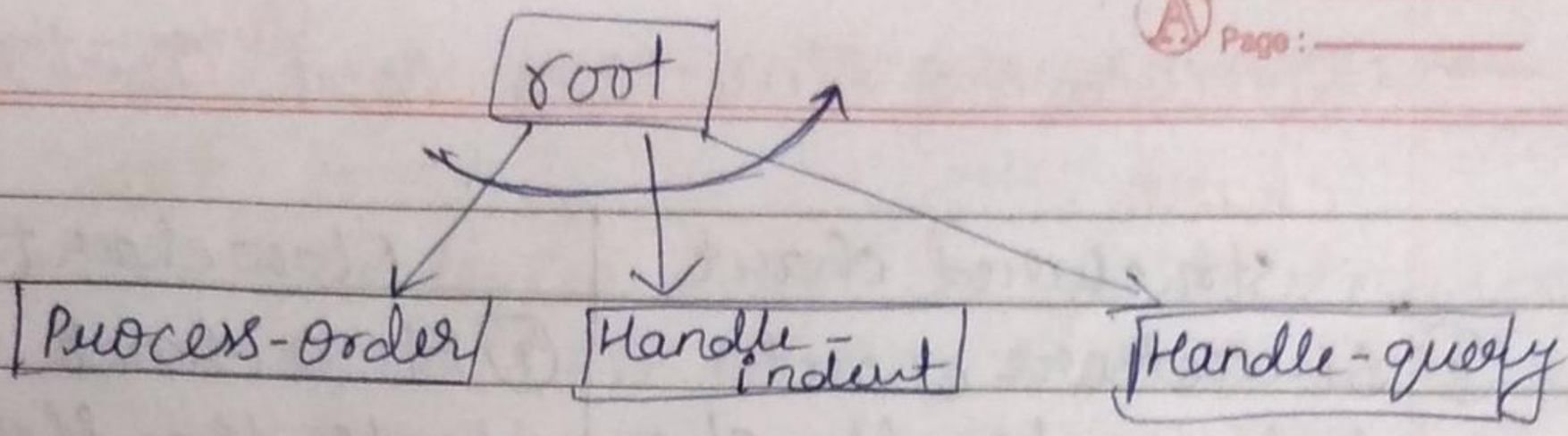
→ Library modules represent frequently called modules.
→ A rectangle with double side edges.



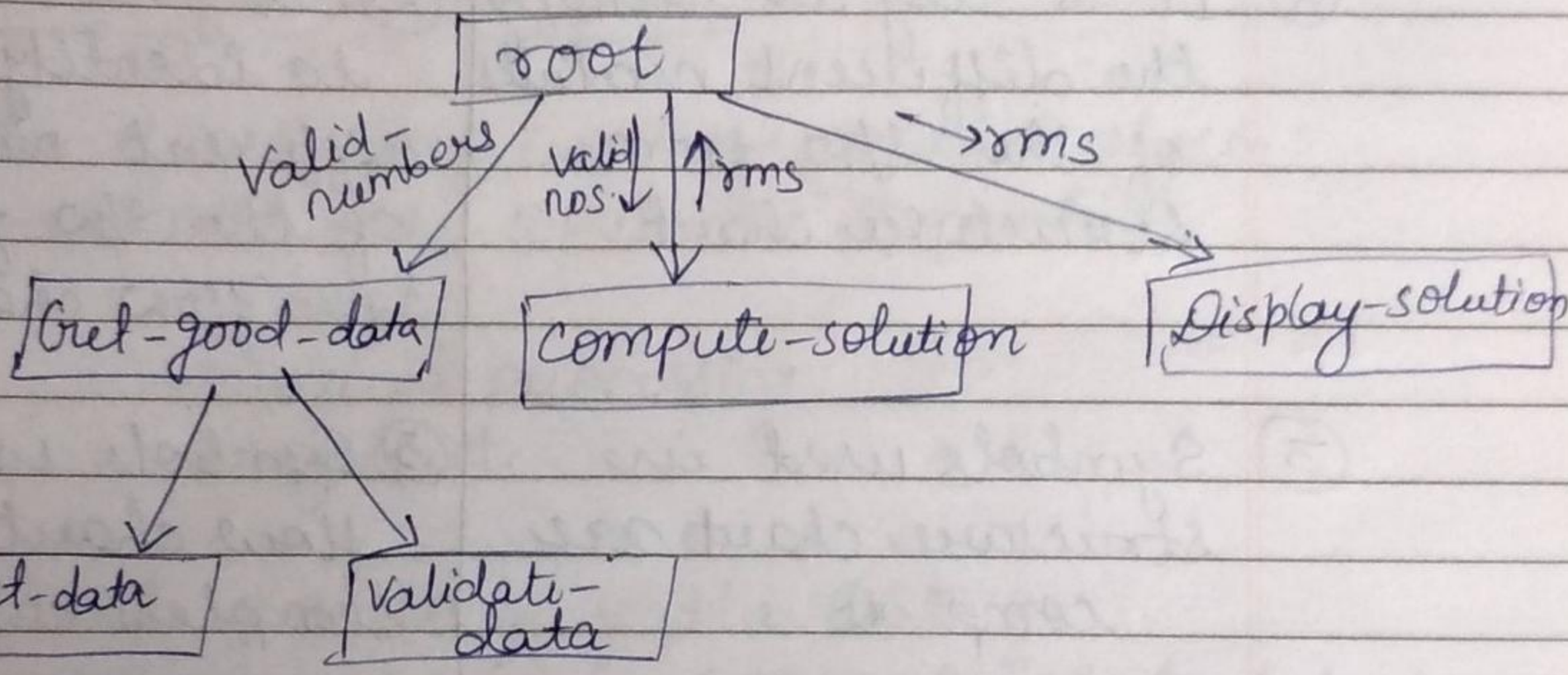
→ Selection
→ represented by diamond symbol



→ Repetition
A loop around control flow arrows denotes that the concerned modules are invoked repeatedly.



e.g. Structured chart (rms)



→ Short Comings of Structure Chart:

→ By looking at a Structure chart:
→ We can not say whether a module calls another module just once or many times.

→ Also, by looking at a Structure chart:
→ We can not tell the order in which the different modules are invoked.

→ Difference b/w Structure chart and Flow chart :—

Structured chart	Flowchart
① Structure chart represents the s/w architecture.	① Flow chart represents the flow of control in program.
② It is easy to identify the different modules of the s/w from structure chart.	② It is difficult to identity the different modules of the s/w from the flow chart.
③ Symbols used in structure chart are complex.	③ Symbols used in flow chart are simple.
④ Data interchange b/w different modules is represented here.	④ Data interchange among different modules is not represented in flow chart.
⑤ Structure chart is complex to construct in comparison of flow chart.	⑤ Flow chart is easier to construct in comparison of structure chart.

Page No.
Date : / /

Transformation of a DFD Model Into Structure Chart :

→ Two strategies exist to guide transformation of a DFD into a structure chart:
  → Transform analysis
  → Transaction Analysis

Transform Analysis

→ The first step in transform analysis:
  → Divide the DFD into 3 parts:
    → input
    → logical processing
    → output

→ input portion in the DFD:
  → processes which convert input data from physical to logical form
  → e.g. read characters from the terminal and store in internal tables or lists.

→ Each input portion called an afferent branch.

→ Output portion of a DFD:
  → transforms output data from logical form to physical form.
  eg. from list or array into output characters.
→ Each output portion called an efferent branch.

→ The remaining portions of a DFD called central transform.

→ Derive structure chart by drawing one functional component per :-
- the central transform
- each afferent branch
- each efferent branch

→ First level of structure chart:
- Draw a box for each input and output units.
- A box for the central transform.

→ Next, refine the structure chart:
- Add subfunctions required by each high-level module.
- Many levels of modules may required to be added.

** Factoring:

- The process of breaking functional components into subcomponents.

→ Factoring includes adding:
- Read and write modules,
- Error-handling modules,
- Initialization and termination modules, etc.

→ Finally check:
- whether all bubbles have been mapped to modules.

** Transaction Analysis :-

→ Useful for designing transaction processing programs.

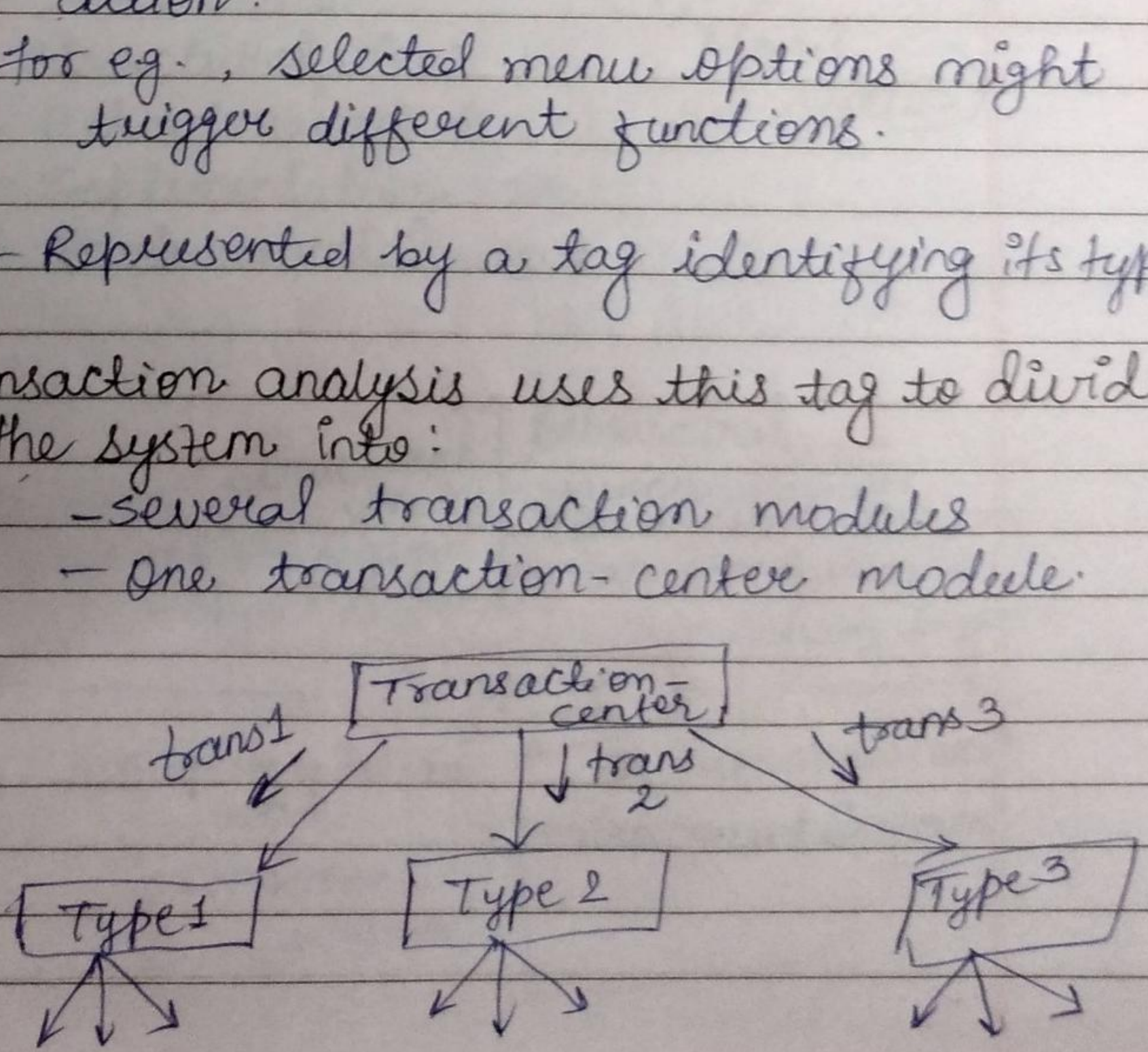
- Transform-centered systems:
  - characterized by similar processing steps for every data item processed by input, process, and output bubbles.

- Transaction-driven systems:
  - One of several possible paths through the DFD is traversed depending upon the input data value.

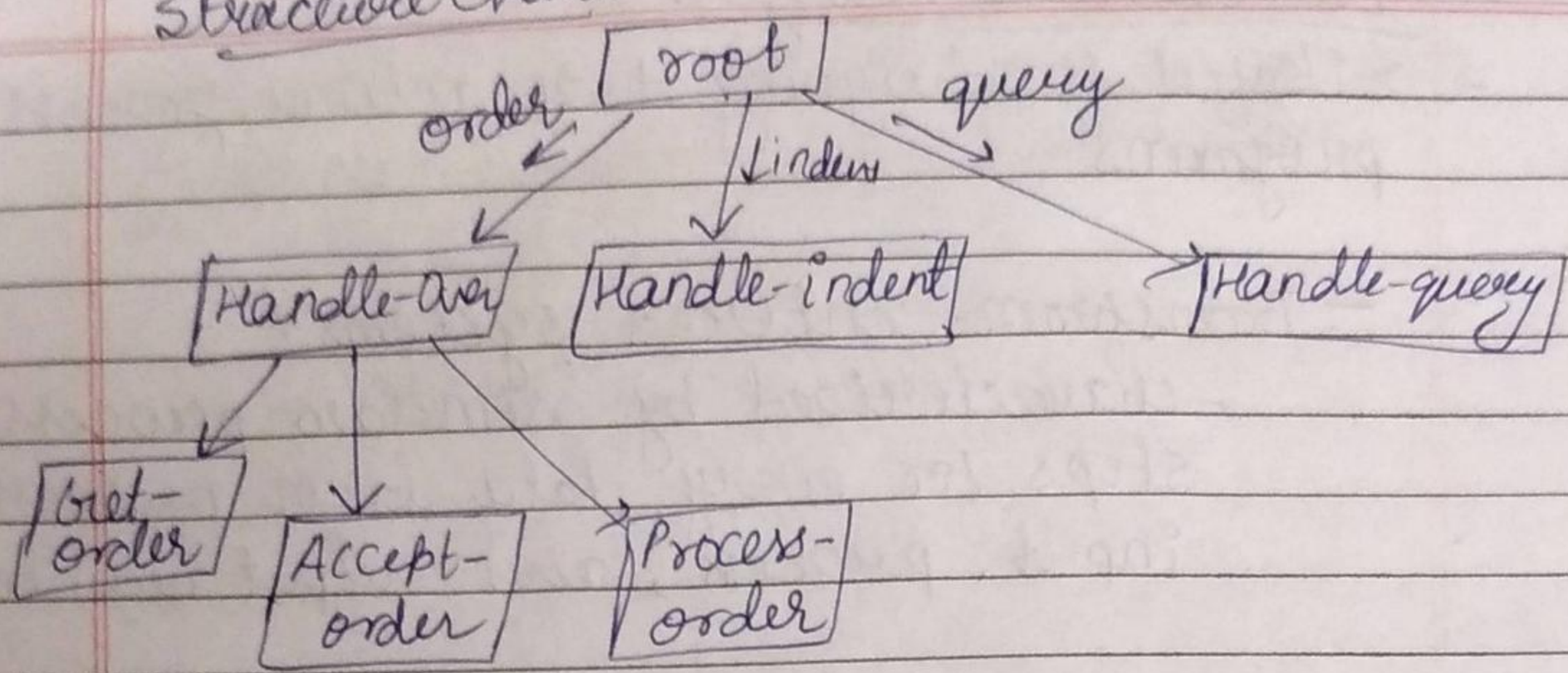
→ Transaction:
- Any input data value that triggers an action:
  for eg., selected menu options might trigger different functions.

- Represented by a tag identifying its type.

→ Transaction analysis uses this tag to divide the system into:
  - several transaction modules
  - One transaction-center module.



Structure Chart



→ Object Oriented Design

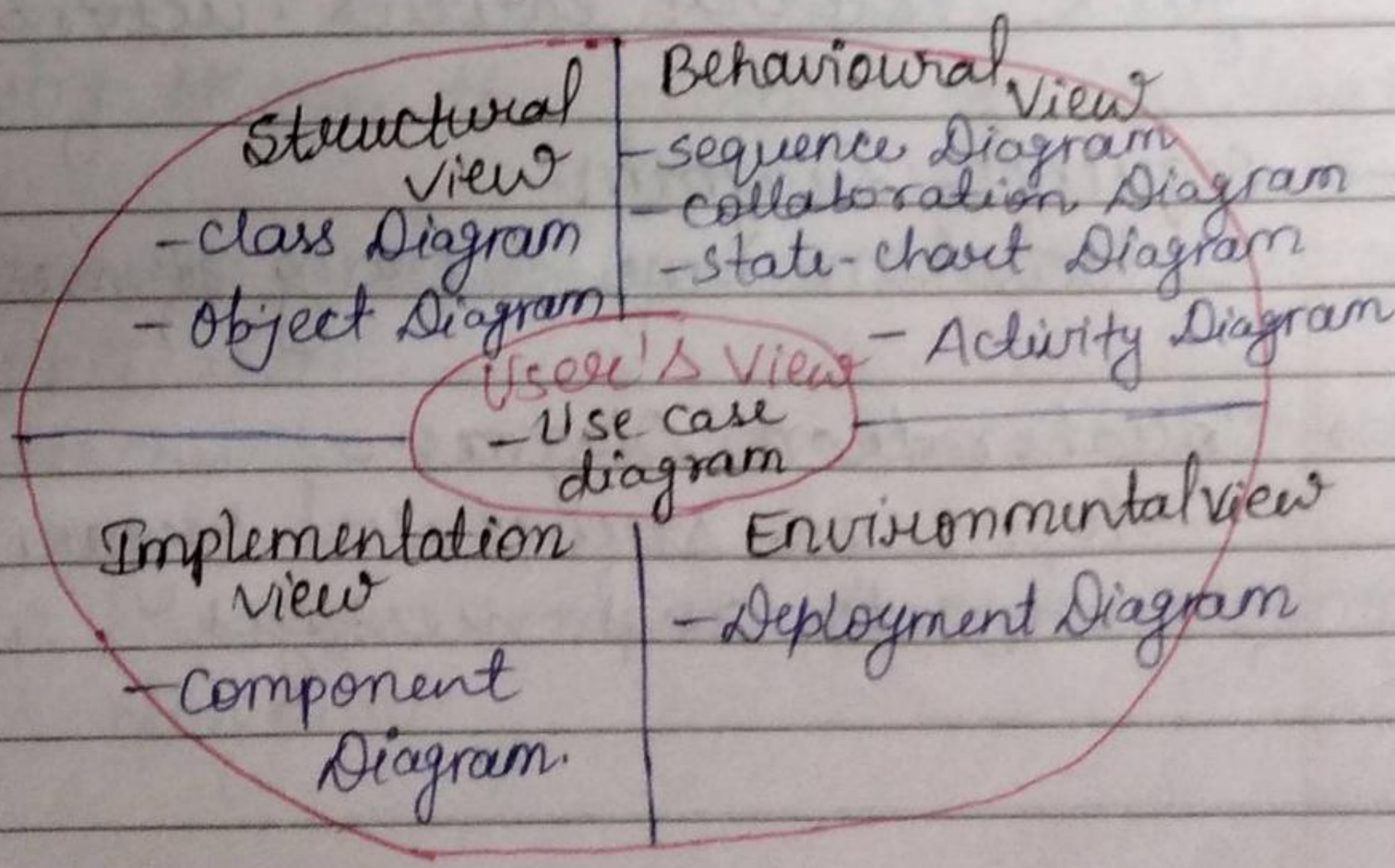
Object Modelling using UML

→ unified Modelling language

→ UML is a modelling language.
  → Not a system design or development Methodology.
→ Used to document object-oriented analysis and design results.
→ Independent of any specific design methodology.

→ **Views:**
Provide different ~~peep~~ perspective of a S/W system.

→ Views of a system:- (UML Model Views)
- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view



Structural Diagrams

→ Class Diagram
- Set of classes and their relationships

→ Object Diagram
- Set of objects (class instances) and their relationships

→ Component Design Diagram
- logical groupings of elements and their relationships

→ Deployment Diagram
- set of computational resources (nodes) that host each component

Behavioral Diagrams

→ Use case Diagram
- high-level behaviors of the system, user goals, external entities: actors

→ Sequence Diagram
- focus on time ordering of messages

→ Collaboration Diagram
- focus on structural organization of objects and messages

Lec-31

Date :
Page :

→ State chart Diagram
— event driven state changes of system.

→ Activity Diagram
— flow of control b/w activities.

→ Class Diagram

• Classes: → (Template for object creation)
→ considered as abstract data type
— Entities with common features, i.e. attributes and operations.
— Represented as solid outline rectangle with compartments.
— Compartments for name, attributes, and operations.

→ The Class diagram depicts a static view of an application. It represents the types of objects residing in the system and relationships b/w them.

→ Following ~~it~~ are the purpose of class diagrams:

1. It analyses and designs a static view of an application.
2. It describes the major responsibilities of a system.
3. It is base for component and deployment diagrams.

→ Benefits of Class Diagrams:

1. It can represent the object model for complex systems.

2. It reduces the maintenance time by providing an overview of how an application is structured before coding.

3. It provides a general schematic of an application for better understanding.

→ What are the different types of Relationships Among classes?

Four types of relationships:

- Inheritance
- Association
- Aggregation/composition
- Dependency

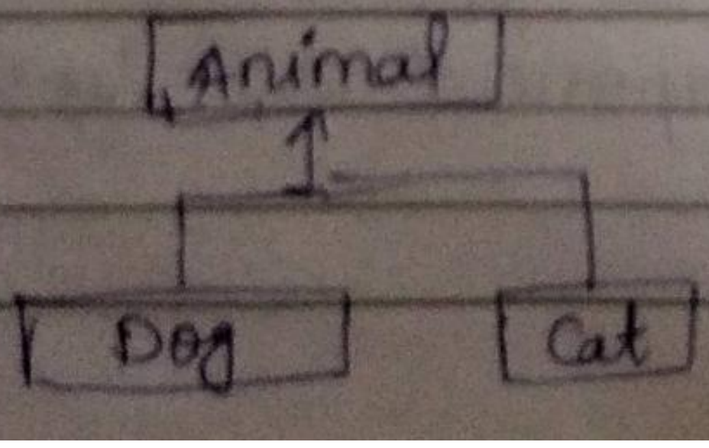
→ Inheritance

→ Allows to define a new class (derived class) by extending an existing class (base class).

- Represents generalization-specialization

- Allows redefinition of the existing methods (method overriding).

e.g.



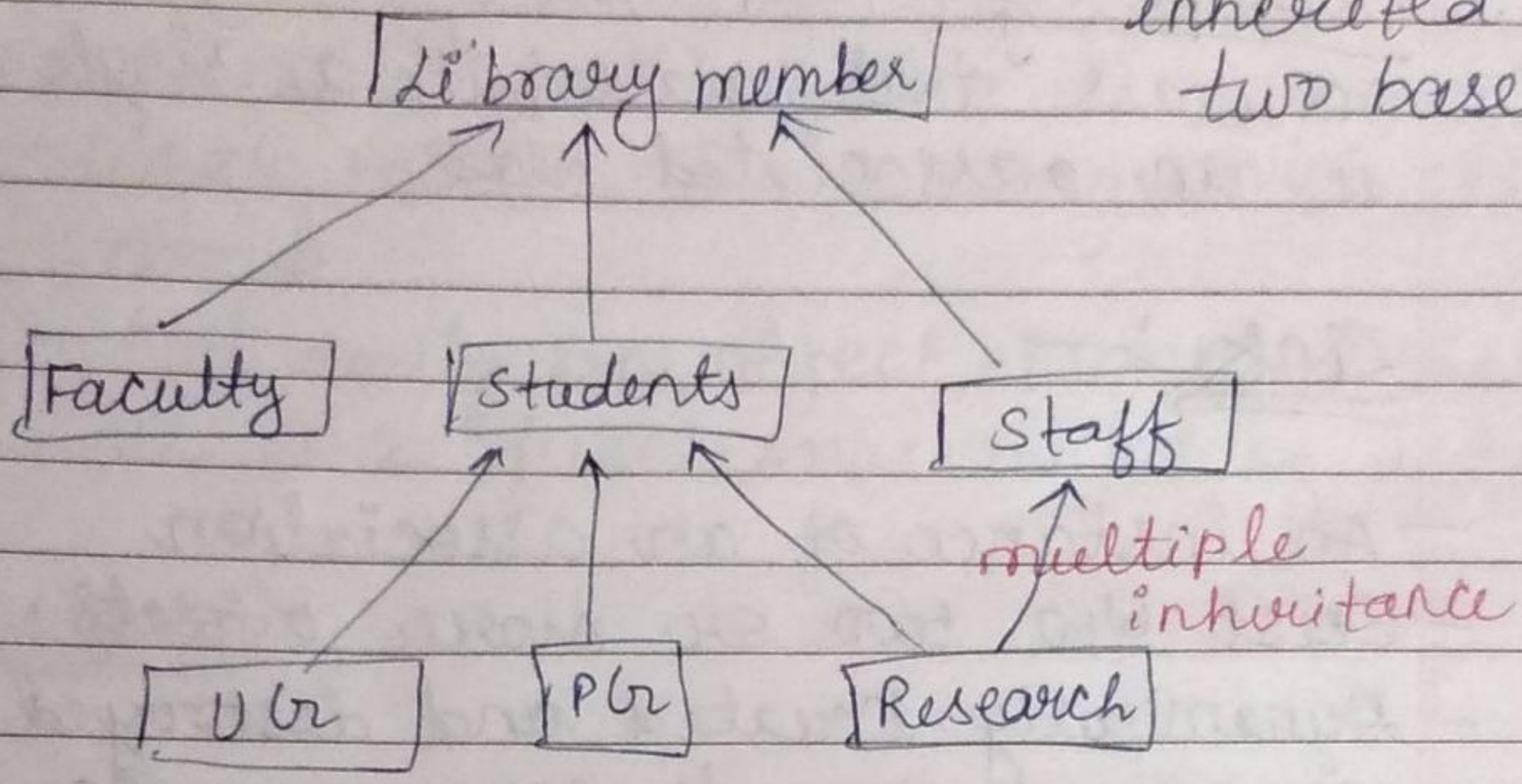
"A Dog ISA Animal"
"A Cat ISA Animal"

→ An attribute ~~of~~ of a class represents a characteristic of a class.

Date : / /
Page :

→ Multiple Inheritance :-

A derived class may inherited from two base classes.



→ Generalization :- Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. ~~o~~ Shared characteristics can be attributes, associations, or methods.

→ Specialization means creating new subclasses from an existing class.

→ Association :-

- An association represents a relationship between two classes:
- Enables objects to communicate with each other:
- Thus one object must "know" the address of the corresponding object in the association.
- Usually binary:
  - But in general can be n-ary.

Date :
Page :

Multiplicity : The number of objects from one class that relate with a single object in an associated class.

Link :-

- An instance of an association
- Exist b/w two or more objects
- Dynamically created and destroyed as the run of a system proceeds.

e.g. An employee joins an organization, leaves that organization and joins a new organization etc.

Date : / /
Page :

→ Use Case Model :

→ Consists of a set of "use cases"
→ It is the central model :
- Other models must conform to this model.
- Not really an object-oriented model, it is a functional model of a system.

→ Use case

The use case is made up of a set of possible sequences of interactions b/w systems and users in a particular environment and related to a particular goal.

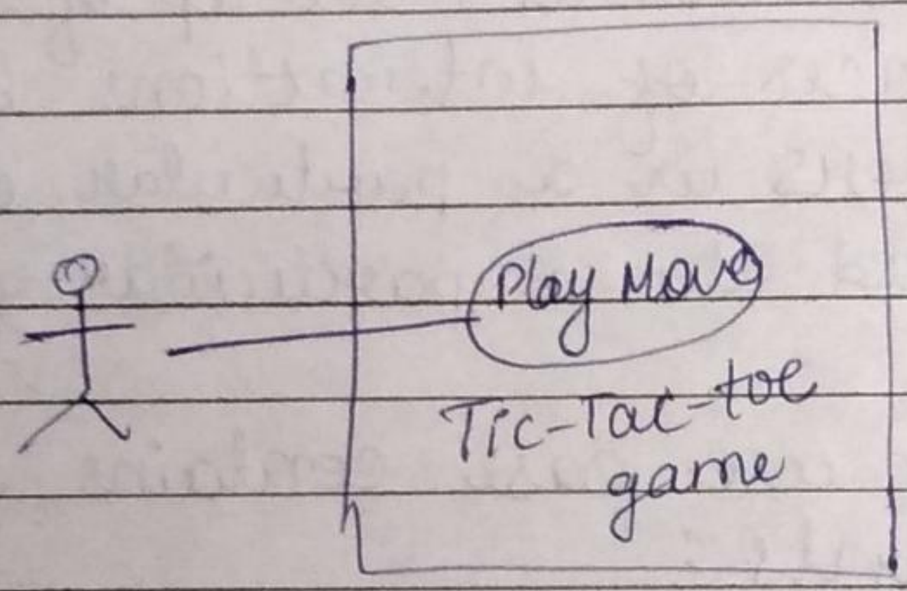
Every use case contains three essential elements :
→ The actor : The system user - this can be a single person or a group of people interaction with the process.

→ The goal : The final successful outcome that completes the process.

→ The system : The process and steps taken to reach the end goal, including the necessary functional requirements and their anticipated behaviours.

→ Representation of Use cases :-

→ Represented in a use case diagram
→ A use case is represented by an ellipse.
→ System boundary is represented by a rectangle
→ Users are represented by stick person icons.
→ Communication relationship b/w actor and use case by a line.



Factoring Use Cases :-

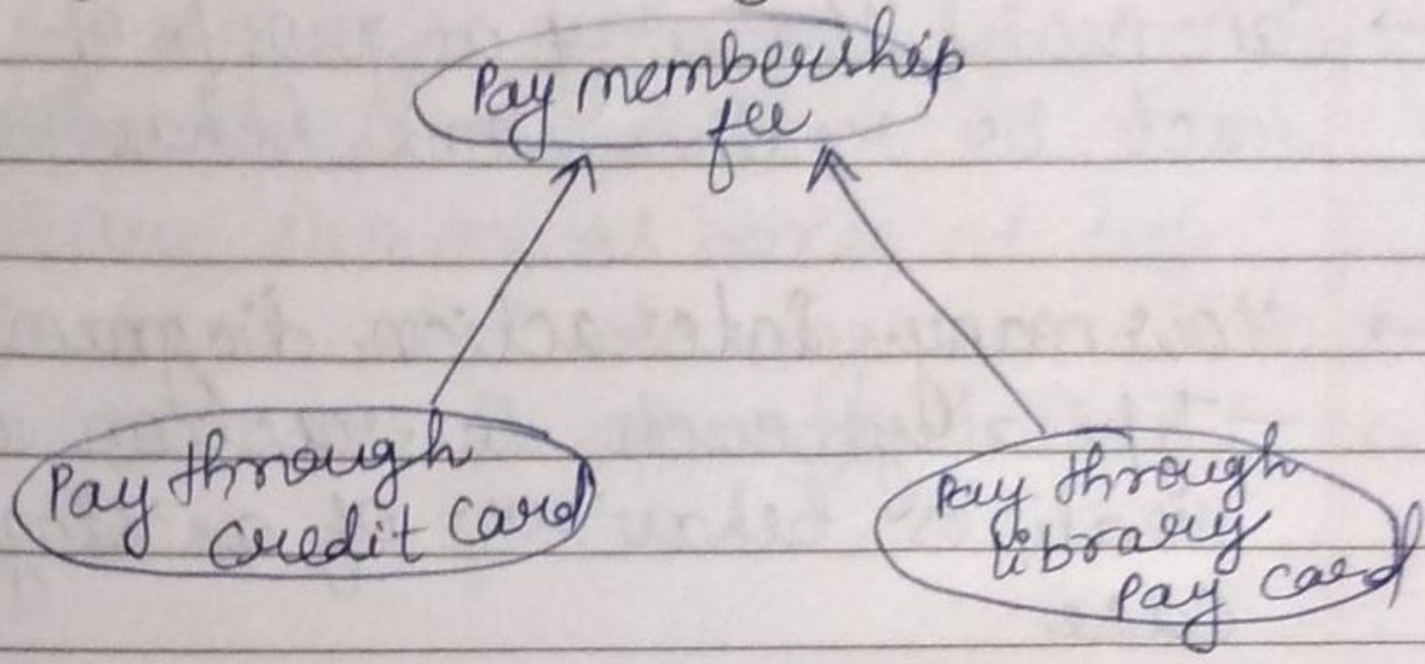
→ Two main reasons for factoring :

- Complex use cases need to be factored into simpler use cases.
- Helps represent common behavior across different use cases.

→ Three ways of factoring :
- Generalization
- Include
- Extend

Date :
Page :

→ Factoring Use cases Using Generalization :-



→ sequence diag.
→ collaboration diag.

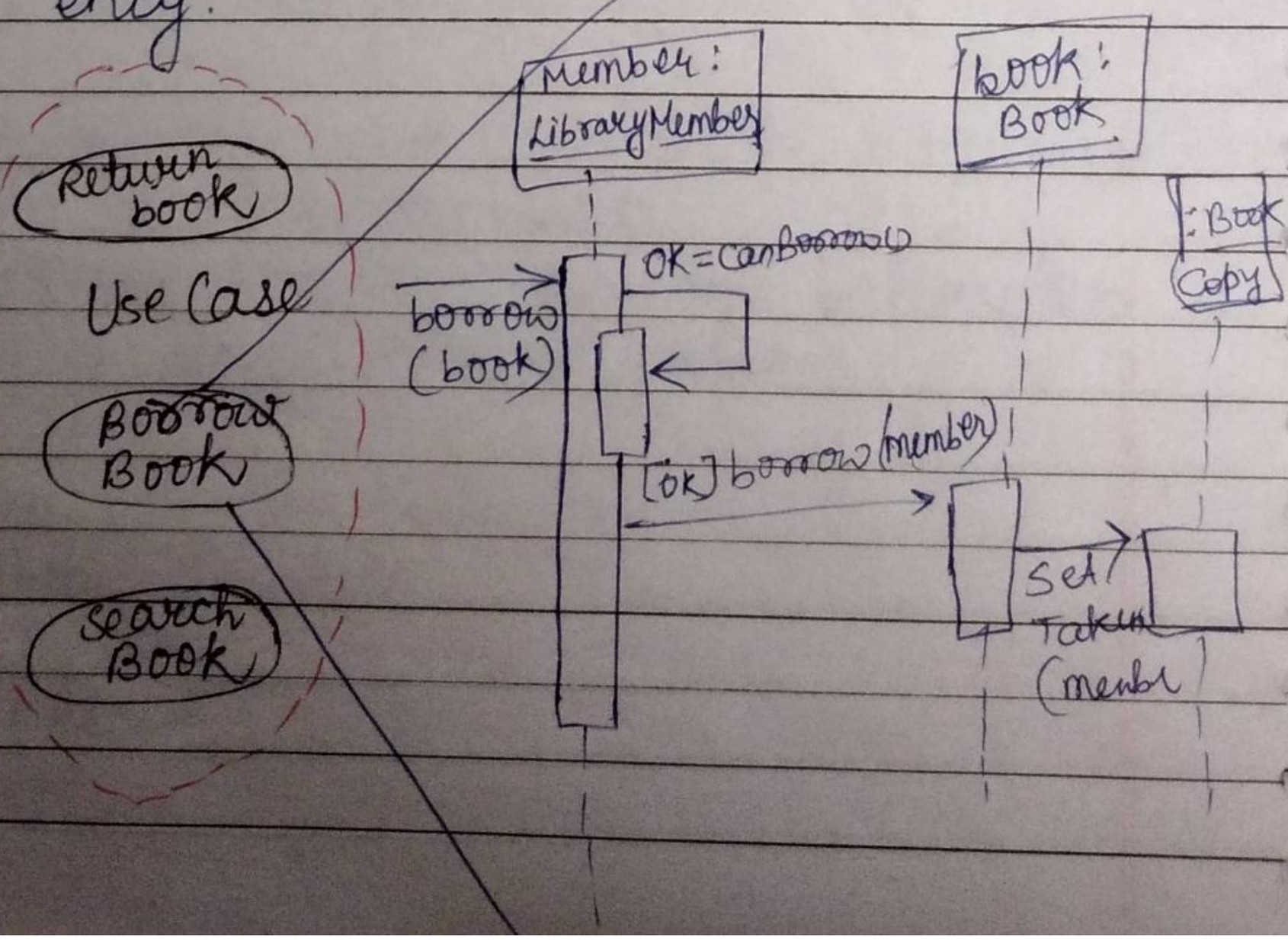
Interaction Diagram (behavioural view)

→ Can model the way a group of objects interact to realize some behaviour.

→ How many interaction diagrams to draw?
— Typically each interaction diagram realizes behaviour of a single use case.
— Draw one sequence diagram for each use case.

→ Captures how objects interact with each other:
— To realize some behaviour (use case execution)

— Emphasizes time ordering of messages.
→ Can model:
— Simple sequential flow, branching, iteration, recursion, and concurrency.



Date :
Page :

→ Sequence Diagram

→ Shows interaction among objects in a two-dimensional chart.

→ Objects are shown as boxes at top.

→ If object created during execution:
  — Then shown at appropriate place in time line

→ Object existence is shown as dashed line (lifeline).

→ Object activeness, shown as a rectangle on lifeline.

Notes:

→ A base class is said to be a generalisation of its derived classes.

→ Each derived class can be considered as a specialisation of its base case. because it modifies or extends the basic properties of the base class in certain ways.

→ Redefinition in a derived class of a method that already exists in its base class is termed as method overriding.

Abstract class:
Classes that are not intended to produce instances of themselves are called abstract classes.
In other words, an abstract class cannot be instantiated into objects.

What is the use of defining an abstract class?
Abstract classes merely exist so that behaviour common to a variety of classes can be factored into one common location, where they can be defined once.

* Abstraction:
The main purpose of using the abstraction mechanism is to consider only those aspects of the problem that are relevant

to a ~~problem~~ given purpose and to suppress all aspects of the problem that are not relevant.

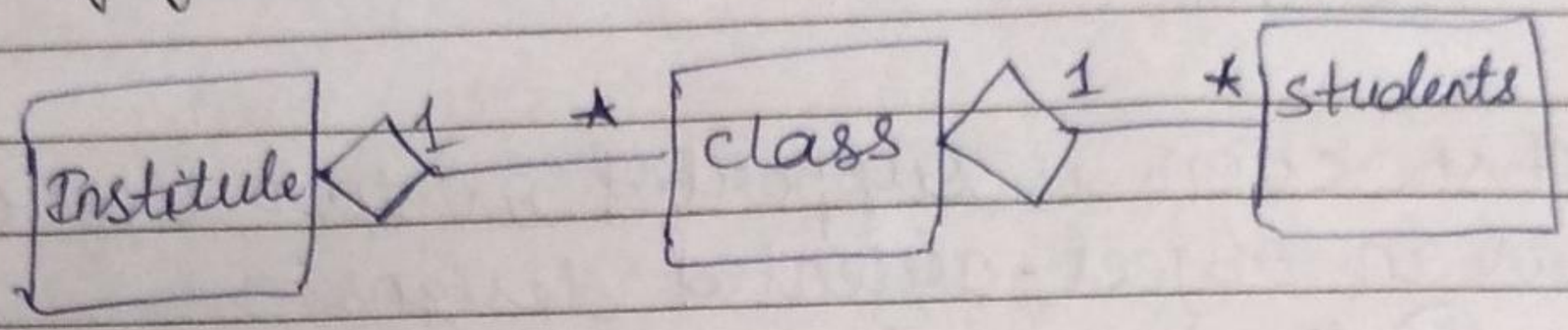
→ Abstraction is supported in two different ways in an object-oriented designs:-

(i) Feature abstraction
(ii) Data abstraction

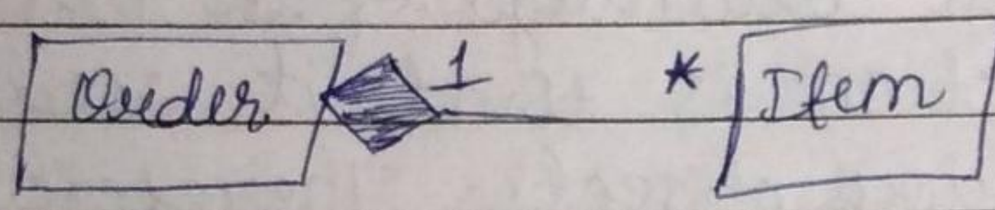
→ An important advantage of the principle of data abstraction is that it reduces coupling among various objects. Therefore it leads to a reduction of the overall complexity of a design, and helps in easy maintenance and code reuse

Note contd.. from Sir's Pdf [OOPS and UML]

→ Aggregation Relationship



→ Composition Relationship
Life of item is same as the order.



→ Aggregation:

- An aggregate object contains other objects.
- Aggregation limited to tree hierarchy:
  - No circular inclusion relation.

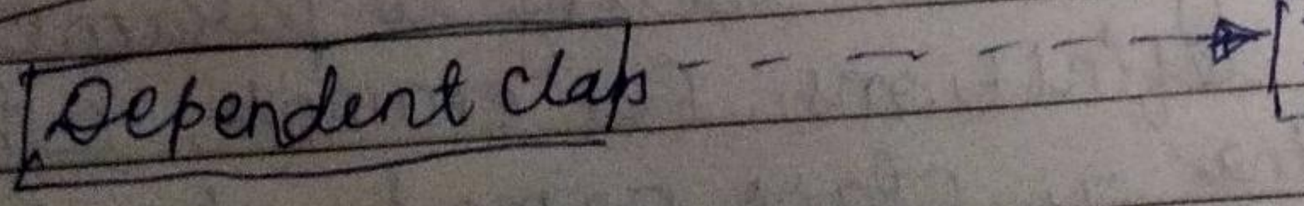
⇒ Association Vs. Composition

Composition and aggregation are two types of association which is used to represent relationships b/w two classes.

Diff.

Key	Composition	Aggregation
1. Basic	Composition (mixture) is a way to wrap simple objects or data types into a single unit.	Aggregation (collection) differs from ordinary composition in that it does not imply ownership.
2. Relationship	In composition, parent entity owns child entity.	In Aggregation, parent has-A relationship with child entity.
3. UML Notation	It is denoted by a filled diamond.	It is denoted by an empty diamond.
4. Life Cycle	child doesn't have their own life time	child can have their own life time
5. Association	It is a strong association.	It is a weak association.

Class Dependency



+ PERT chart is a project management tool used to schedule, organize, and coordinate tasks within a project. It provides a graphical representation of the project timeline, allowing for easy identification of task dependencies and critical paths.

* Polymorphism
- Denotes poly (many) morphism (forms).
- Under different situations :
  - same message to the same object can result in different actions:
    - Static binding
    - Dynamic binding

- Polymorphism denotes that an object may respond (behave) very differently even when the same operation is ~~ind~~ invoked.

* Static & Polymorphism :-
It occurs when multiple methods in a class implement the same operation through multiple methods having the same method name but different parameter types.

* Dynamic polymorphism :-
It is also called dynamic binding. In dynamic binding, the exact method that would be invoked on a method call is determined at the run time and cannot be determined at compile time.

* Genericity

Genericity is the ability to parameterise class definitions. For example, while defining a class named stack, we

may notice that we need stacks of different types of elements such as integer stack, character stack, floating-point stack etc.

→ Advantages and Disadvantages of OOD :-

Advantages of OOD

→ Code and design reuse
→ Increased productivity
→ Ease of testing and maintenance
→ Better understandability
→ Elegant design:
  - Loosely coupled, highly cohesive objects.
  - Essential for solving large problems.

Disadvantages of OOD

(9) The principles of abstraction, data hiding, inheritance, etc. do incur run time overhead due to the additional code that gets generated on account of these features. This causes an object-oriented program to run a little slower than an equivalent procedural program.